CS345H: Programming Languages

Lecture 3: Lexical Analysis

Thomas Dillig

Syntactic Analysis

- Main Question: How to give structure to strings
- Analogy: Understanding an English sentence
  - First, we separate a string into words
  - Second, we understand sentence structure by diagramming the sentence
- Separating a string into words is called lexing and our topic today
- Observe that this is not necessarily trivial
- Consider the string `if x <> 3 then ...`

Outline

- Informal sketch of lexical analysis
- Issues in lexical analysis
  - Lookahead
  - Ambiguities
- Specifying Lexers
  - Regular Expressions
  - Examples of regular expressions

Lexical Analysis

- Consider the following L program:

```
if x <> y then
   3
else
   "hello"
```

- This "program" is just a string of characters
  `if x <> y then\n\t3\nelse\n\t"hello"`
- Goal: Portion the input string into substrings where the substrings are tokens

What is a Token?

- Token is a syntactic category
- Example in English: noun, verbs, adjectives, ...
- In a programming language: constants, identifiers, keywords, whitespaces...

Tokens

- Tokens correspond to sets of strings
- Identifier in L: strings of letters, digits and '_' starting with a letter
- Integer in L: a non-empty string of digits
- Keywords in L: "let", "lambda", "if", ...
- Whitespace: a non-empty sequence of blanks, newlines, and tabs

What are Tokens For?

- Classify program substrings according to their role
- Output of lexical analysis is a stream of tokens...
- ...which is input to the parser
- Parser relies on token distinction
  - An identifier is treated different than a keyword

Defining a Lexical Analyzer: Step 1

- Define a finite set of tokens
- Tokens describe all items of interest
  - This means no tokens for items not of interest, such as comments, whitespaces,...
- Choice of tokens depends on language and design of the parser

Example

- Recall:
  `if x <> y then\n\t3\nelse\n\t"hello"`
- Useful tokens for this expression: Identifier, Keyword, Integer, Relation, Whitespace,...
- Important point: < is a character here, but token is <>

Defining a Lexical Analyzer: Step 2

- Describe which strings belong to each token
- Recall:
  - Identifier in L: strings of letters, digits and '_' starting with a letter
  - Integer in L: a non-empty string of digits
  - Keywords in L: "let", "lambda", "if", ...
  - Whitespace: a non-empty sequence of blanks, newlines, and tabs

Lexical Analyzer: Implementation

- A lexer implementation must do two things:
  1. Recognize substrings corresponding to tokens
  2. Return the value (called lexeme) of the token
- The lexeme is just the substring
- Example: "234" Token: Integer Lexeme: 234

Example

- Recall:
  `if x <> y then\n\t3\nelse\n\t"hello"`

Lexical Analyzer: Implementation

- The lexer usually discards "uninteresting" toekens that don't contribute to parsing
- Examples: Comments, whitespaces

True Crimes of Lexical Analysis

- Is this as easy as it sounds?
- Not quite!
- Let's look at some history...

Lexical Analysis in FORTRAN

- FORTRAN rule: Whitespace is insignificant
- Example: `VAR1` is the same as `VA R1`
- Reason: Easy to mess up whitespace when typing punch cards
- A terrible design!

Example

- Consider
  ```
  DO 5 I=1,25
  DO 5 I=1.25
  ```


Lexical Analysis in FORTRAN (Cont.)

- Two important points to take away from this example:
  1. The goal is to partition the string. This is implemented by reading left-to-right, recognizing one token at a time
  2. Lookahead may be required to decide where one token ends and the next token begins

Lookahead

- Even our simple example has lookahead issues:
- `i` vs. `if`
- `<` vs. `<>`

Lexical Analysis in PL/I

- PL/I keywords are not reserved
- This means the following is a legal PL/I program
  `IF ELSE THEN THEN = ELSE; ELSE ELSE = THEN`

Lexical Analysis in PL/I (cont.)

- PL/I array references: `array(i)`
- PL/I declarations: `DECLARE(ARG1,...,ARGN)`
- Cannot tell whether `DECLARE` is a keyword or array reference until after the), requiring arbitrary lookahead!
- Notice: PL/I will continue to entertain us throughout this course

Lexical Analysis in C++

- Unfortunately, these problems still exist in today's languages
- C++ template syntax: `vector<int>`
- C++ stream syntax: `cin >> var`
- But there is a conflict with nested templates:
  `list<vector<int>>`

Review

- The goal of lexical analysis is:
  - Partition the input string into lexemes
  - Identify the token of each lexeme
- Left-to-right scan $\Rightarrow$ lookahead sometimes required

Next

- We still need a way to describe the (often infinite) set of lexemes of each token
- And a way to resolve ambiguities
  - Is `if` to variables `i` and `j`?
  - Is <> to operators < and >?

Regular Languages

- We could specify tokens in many ways
- Regular Languages are the most popular
  - Simple and useful theory
  - Easy to understand
  - Efficient to implement

Languages

- Definition: Let Σ be a set of characters, A language over Σ is a set of strings from characters drawn from Σ

Examples of Languages

- Alphabet: English characters Language: English sentences
- Alphabet: Not every string of English characters is an English sentence
- Alphabet: ASCII Language: C programs
- Observe: ASCII character set is different from English character set

Notation

- Languages are sets of strings
- Need some notation for specifying which sets we want
- The standard notation for regular languages is regular expressions

Atomic Regular Expressions

- Single character: $'c' = \{``c"\}$
- Epsilon: $\varepsilon = \{``"\}$

Compound Regular Expressions

- Union: $A + B = \{s | s \in A \text{ or } s \in B\}$
- Concatenation: $AB = \{ab | a \in A \text{ and } b \in B\}$
- Iteration: $A^* = \bigcup_{i \geq 0} A^i$ where $A^i = A...i$ times A

Regular Expressions

- The regular expressions over Σ are the smallest set of expressions including
- ε
- $'c'$ where $c \in \Sigma$
- $A + B$ where A, B are regular expressions over Σ
- AB where A, B are regular expressions over Σ
- A^* where A is a regular expression over Σ
- Regular expressions are simple, but very useful

Example: Keyword

- Keywords: `lambda`, `else`, `if`, ...
- Regular Expression: `'lambda' + 'else' + 'if'+ ...`
- Note: `'lambda'` short for `'l' 'a' 'm' 'b' 'd' 'a'`

Example: Integers

- Integer: non-empty string of digits
- `digit = '0' + '1' + '2' + '3' + '4' + '5' + '6' + '7' + '8' + '9'`
- `integer = digit digit`*
- Abbreviation: $A^+ = AA^*$

Example: Identifier

- Identifier: strings of letters or digits, starting with a letter
- `letter = 'A'+...+'Z'+'a'+...'z'+'_'`
- `identifier = letter (letter + digit)`*
- Question: Is `(letter`* `+ digit`*`)` the same?

Example: Whitespace

- Whitespace: a non-empty sequence of blanks, newlines and tabs
- `(' ' + '\n' + '\t')`$^+$

Example: Phone numbers

- Regular expressions are everywhere!
- Consider (757)-221-1234
  - $\Sigma = \texttt{digits} \cup \{-, (,)\}$
  - `exchange = digit`^{3}
  - `phone = digit`^{4}
  - `area = digit`^{3}
  - `phone_number = '(' area ')-'exchange'-'phone`

Last Example: email addresses

- Consider W&M cs emails: `anyone@cs.wm.edu` format
- Σ = `letters` $\cup$ `{ . , @}`
- `name = letter`$^+$
- `address = name '@' name '.' name '.' name`

Other real-world examples

- File names
- Grep tool
- Anything else?

Summary

- Regular expressions describe many useful languages
- Regular languages are only a specification, we still need an implementation
- Next time: Given a string s and a regular expression R is $s \in L(R)$?